Bent Function Discovery by Reconfigurable Computer

Jon T. Butler
Department of Electrical and Computer Engineering
Naval Postgraduate School
Monterey, CA 93943 U.S.A.
jon_butler@msn.com

Abstract

Bent Boolean functions are important in the encoding/decoding of secure messages. Because they are the most nonlinear of all functions, they are the least susceptible to linear attack. However, bent functions are rare and difficult to discover. The only known way to enumerate all bent functions is by a sieve in which many prospective functions are tested.

This is a tutorial description of the process of bent Boolean function discovery by a reconfigurable computer. Specifically, we discuss the use of SRC Computers' SRC-6 reconfigurable computer in sieving through a large number of functions. We show why this process is much faster than on a conventional computer (up to 60,000 times), and we discuss the circular pipeline as a method to improve the throughput even further. The circular pipeline takes advantage of the fact that most functions pass not even one test for bentness. The improvement in throughput due to the circular pipeline depends on the relationship between distances among functions, but it is approximately 500 times better than our present throughput.

1 Introduction

In ancient times, the encryption of messages was likely to have been as simple as writing on parchment, since few people could read. However, as more people became literate, encryption had to become more sophisticated. A common encryption method that has survived to modern times is substitution – replacing "e" with "p", "t" with "i", ... , for example. A consistent substitution, such as *always* replacing "e" with "p", "t" with "i", ... is relatively easy to decode. With enough ciphertext and patience, one can, by hand, decode the message. You simply observe that certain letters are more frequent than others. For example, in English, "e" occurs about 15% of the time and in German, about 17% of the time, versus about 4% if all letters were equally likely. With a computer, breaking the code becomes trivial, especially if it stores statistics on the occurrence of letters in plaintext, statistics on letters that occur at the beginning of words, statistics on combinations of letters, and dictionaries.

To avoid the vulnerabilities of consistent substitution, cryptologists have used keystreams. Often a binary keystream of (pseudo)random bits is exclusive ORed with the bits encoding letters. In this way, each plaintext letter is encoded as a different cyphertext letter each time.

For more than 50 years, cryptologists have studied the use of Boolean functions in keystreams. Indeed, Shannon [11] discussed the use of *diffusion* and *confusion* in secure communications. Diffusion occurs when a small change in the input text creates a large change in the encoded text. Confusion occurs when the nonlinearity of the system creates a hurdle against decryption using a linear attack.

The most nonlinear of all Boolean functions are the bent functions. Introduced in 1976 by Rothaus [7], bent functions are as far away from linear functions as possible. By "far", we mean in the Hamming distance between their truth tables. However, there exists a paradox. It is easy to identify whether a given function f is bent. One simply applies a distance test. On the other hand, it is an entirely different matter to efficiently generate all bent functions. Some constructions exist for generating bent functions from other bent functions. However, an efficient way to generate *all* bent functions remains to be discovered.

In this paper, we present a sieve process in which many prospective functions are tested for bentness. We show results obtained from this approach. A special feature of this analysis is the use of a reconfigurable computer, the SRC-6. With this approach, we are able to achieve a speedup of 60,000 over a conventional computer.

In the next section, we introduce the background and notation. In Section 3, we discuss the programming of a reconfigurable computer for generating bent functions. Next, we discuss a method to improve

Report Documentation Page

Form Approved
OMB No. 0704-0188

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
SEP 2010		

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Bent Function Discovery by Reconfigurable Computer	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Naval Postgraduate School,Department of Electrical and Computer Engineering,Monterey,CA,93943**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Bent Boolean functions are important in the encoding/decoding of secure messages. Because they are the most nonlinear of all functions, they are the least susceptible to linear attack. However, bent functions are rare and difficult to discover. The only known way to enumerate all bent functions is by a sieve in which many prospective functions are tested. This is a tutorial description of the process of bent Boolean function discovery by a reconfigurable computer. Specifically, we discuss the use of SRC Computers? SRC-6 reconfigurable computer in sieving through a large number of functions. We show why this process is much faster than on a conventional computer (up to 60,000 times), and we discuss the circular pipeline as a method to improve the throughput even further. The circular pipeline takes advantage of the fact that most functions pass not even one test for bentness. The improvement in throughput due to the circular pipeline depends on the relationship between distances among functions, but it is approximately 500 times better than our present throughput.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**12**	

the throughput of the reconfigurable computer, and we show experimental results that suggest the extent to which throughput can be improved. In Section 5, we give analytical results, and in Section 6, we give concluding remarks.

2 Background and Notation

It is convenient to represent the truth table of an n-variable function by a string of 2^n bits. For example, $f_1(x_1, x_2, x_3, x_4) = x_1x_2x_3x_4$ is represented as $TT(f_1) = (0000000000000001)$, and $f_2(x_1, x_2, x_3, x_4) = x_1x_2 \oplus x_3x_4$ is represented as $TT(f_2) = (0001000100011110)$.

It is also convenient to represent an n-variable function in its algebraic normal form. Specifically,

Definition 2.1. *The* **algebraic normal form** *(***ANF***) of a function f is $f = \sum_{\mathbf{a}\in\mathbb{F}_2} c_{\mathbf{a}} x_1^{a_1} x_2^{a_2} \ldots x_n^{a_n}$, where $\sum$ is the exclusive OR, $\mathbf{a} = (a_1, a_2, \ldots, a_n)$, $c_{\mathbf{a}}, a_i \in \mathbb{F}_2$, $x_i^0 = 1$, and $x_i^1 = x_i$. $(c_0\, c_1\, \ldots\, c_{2^n-1})$ is the* **ANF representation** *of f.*

Example 2.1. *$f_1 = x_1x_2x_3x_4$ has the ANF $f = x_1x_2x_3x_4$ and the ANF representation $ANF(f_1) = (0000000000000001)$. $f_2 = x_1x_2 \oplus x_3x_4$ has the ANF $f_2 = x_1x_2 \oplus x_3x_4$ and the ANF representation $ANF(f_2) = (0001000000001000)$. (End of Example)*

The algebraic normal form of a function is also known as the positive polarity Reed-Muller form.

Definition 2.2. *A* **linear** *function is the constant 0 function or the exclusive-OR of one or more variables. An* **affine** *function is a linear function or the complement of a linear function.*

Example 2.2. *There are 16 linear functions on 4 variables, 0, x_1, x_2, x_3, x_4, $x_1 \oplus x_2$, $x_1 \oplus x_3$, $x_1 \oplus x_4$, $x_2 \oplus x_3$, $x_2 \oplus x_4$, $x_3 \oplus x_4$, $x_1 \oplus x_2 \oplus x_3$, $x_1 \oplus x_2 \oplus x_4$, $x_1 \oplus x_3 \oplus x_4$, $x_2 \oplus x_3 \oplus x_4$, and $x_1 \oplus x_2 \oplus x_3 \oplus x_4$. The 4-variable affine functions consists of these functions and their complements. Note that, if $g(x_1, x_2, \ldots x_n)$ is a linear function in ANF, then the ANF of its complement is $1 \oplus g$. (End of Example)*

Definition 2.3. *The* **Hamming distance** *$d(f, g)$ between two functions f and g is the number of places where their truth table representations differ.*

Definition 2.4. *The* **nonlinearity $\mathbf{NL_f}$** *of a function f is the minimum Hamming distance between f and an affine function.*

Example 2.3. *$f = x_1x_2x_3x_4$ has nonlinearity 1, since converting the single 1 to a 0 in its truth table representation creates the truth table representation of the constant 0 function, which is affine. $g = x_1x_2 \oplus x_3x_4$ has a distance 6 or 10 from any affine function. Thus, its nonlinearity is 6. (End of Example)*

Definition 2.5. *Let f be a Boolean function on n-variables, where n is even. f is a* **bent function** *if its nonlinearity is maximum among n-variable functions.*

Example 2.4. *Rothaus [7] showed that bent functions have nonlinearity $2^{n-1} - 2^{\frac{n}{2}-1}$. Thus, $f = x_1x_2x_3x_4$ is not bent ($NL_f = 1$), and $g = x_1x_2 \oplus x_3x_4$ is bent ($NL_g = 6$). (End of Example)*

Definition 2.6. *The* **degree of a product term** *is the number of variables in that term. The* **degree of a function** *f is the maximum of the degrees among the product terms in the ANF of f.*

Example 2.5. *$f = x_1x_2x_3x_4$ has degree 4 and $g = x_1x_2 \oplus x_3x_4$ has degree 2. (End of Example)*

Definition 2.7. *Functions f and h belong to the same* **affine class** *if and only if $f = h \oplus a$, where a is an affine function.*

Example 2.6. *$f = x_1x_2x_3x_4$, a non-bent function, belongs to an affine class of 32 functions. $g = x_1x_2 \oplus x_3x_4$, a bent function, belongs to an affine class of 32 functions. (End of Example)*

Each affine class contains the same number of functions, namely 2^{n+1}. Also, all functions in the same affine class as a non-affine function f have the same degree. This is because the ANF's of two functions in the same affine class differ only in the linear and constant coefficients. This same statement is *not* true of the affine class of affine functions. In this class, the nontrivial functions have degree 1, while the two trivial (constant) functions have degree 0.

3 Programming the SRC-6 Reconfigurable Computer to Sieve Bent Functions

The SRC-6 reconfigurable computer is a product of SRC Computers, which was formed by **S**eymour **R**. **C**ray in 1995. It has two Intel Pentium Xeon microprocessors through which the user interfaces with the FPGAs. There are 10 FPGAs, four Xilinx Virtex-II XC2V6000 and six Virtex-Pro XC2VP100 FPGAs. These are high-end FPGAs with more than 6 million transistors each. All run at a fixed clock frequency of 100 MHz. To access the machine, one logs onto one of the two microprocessors, compiles the code (some of which runs on the microprocessor and some on the FPGA(s)), and then executes it. During execution, both the microprocessor and FPGAs are executing. For the sieving of bent function, we used three programs 1) C code that runs on the microprocessor and initiates the process and prints/stores the data obtained, 2) C code that runs on the FPGA and enumerates the functions under test, and 3) Verilog code that runs on the FPGA and does most of the computation. To ease the burden on the programmer, SRC Computers has supplied many system macros for use in C programs. This has helped to eliminate the need for Verilog code. However, we found that Verilog code was an absolute necessity. Fig. 1 from [14] shows the SRC-6's compilation process, including place and route. Here, the box labeled "μProc Compiler" corresponds to code that is compiled into machine code that runs on the microprocessor, while the box labeled "MAP Compiler" corresponds to code that is compiled into circuits on the FPGA. This includes both C code and Verilog. Verilog code, in our version of the SRC-6, is compiled using Synplify Pro®.

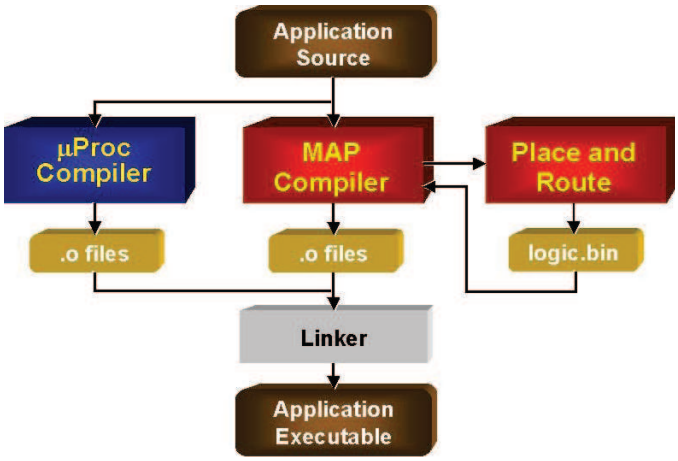

Figure 1: The SRC-6's compilation process [14].

Fig. 2 shows a circuit that enumerates all functions and computes their nonlinearity. On the left is a counter that runs through all truth tables. This is described in C, and runs on the FPGA. To its right, the circuit labeled "Nonlinearity Computation" is described in Verilog and runs on the FPGA. To its right is an oval labeled "Update NL Counter". This data collection part is done on the microprocessor. The data it collects is exported to either Excel or MATLAB for display.

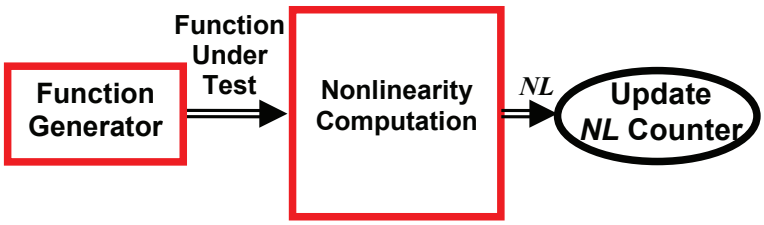

Figure 2: The nonlinearity enumeration circuit.

Fig. 3 shows the architecture of **Nonlinearity Computation** circuit of Fig. 2. **Function** shown on the

left is applied to a circuit consisting of a set of $m = 2^n$ bitwise exclusive OR gates, whose other input is an affine function. The output of each is a distance vector between **Function** and an affine function. The number of 1's in each distance vector is the distance the function under test is from that affine function. The circuit **Ones Count** counts the 1's and produces a binary number that is this count. This circuit was shown by Komamiya [5] (and reformulated by Sasao [13]) to have an elegant form as basic symmetric functions expressed as the exclusive-OR of product terms. We use 4-input LUTs and binary adders to realize this circuit, which is easily scaled across n, where n is the number of variables. Simple ripple-carry adders are highly optimized for speed in the Xilinx Virtex-Pro FPGA, and can be easily adapted to any reasonable wordwidth.

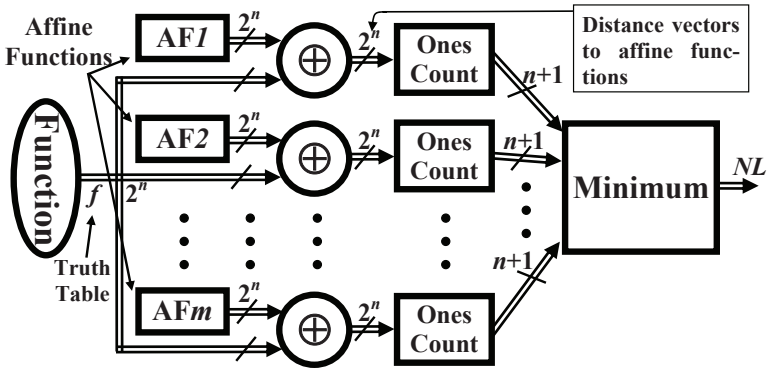

Figure 3: The architecture of the Nonlinearity Computation circuit [10].

Fig. 4 shows the **Ones Count** circuit of Fig. 3. The 4-input LUTs are shown on the left side. A set of three of these forms a 3-bit binary number that is 000, 001, 010, 011, and 100, corresponding to 0, 1, 2, 3, or 4 1's in the four inputs. These are applied to a tree of adders whose output is a binary number representing the number of 1's in the distance vector.

Fig. 5 shows the **Minimum** Circuit of Fig. 3. Like the **Ones Count** circuit, it is a tree. In this case, the **Minimum** circuit consists of 2-input 1-output comparator circuits in which the output is the minimum of the two inputs.

Note that the Nonlinearity Computation circuit is combinational logic. Unfortunately, its delay exceeds 10 nsec., which is the period of the SRC-6's 100 MHz clock. Therefore, in order to process one function in each clock cycle, it is necessary to pipeline the Nonlinearity Computation circuit so that the delay in each stage is 10 nsec. or less. For example, in computing the bent functions for $n = 6$, the Nonlinearity Circuit needed eight stages. Since many prospective functions must be tested, the additional latency due to the pipeline has little effect on the time of execution. Indeed, we ignore it in determining the time of execution. Table 1 from [10] compares the computation times required by an FPGA on the SRC-6 reconfigurable computer with the computation times required by microprocessor used in the SRC-6. The latter is an Intel Xeon microprocessor running at 2.8 GHz. Table 1 shows n in the first

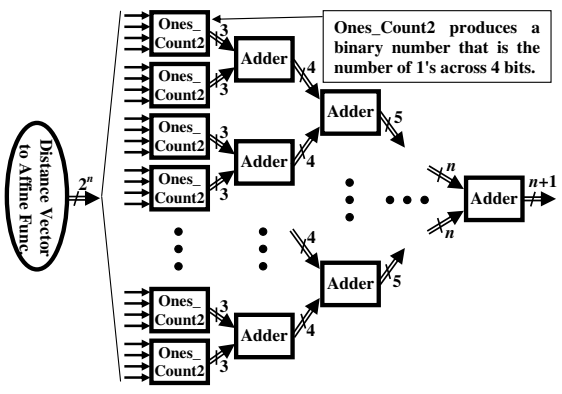

Figure 4: The architecture of the **Ones Count** circuit [10].

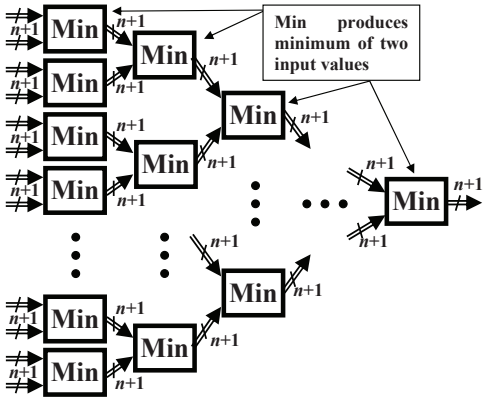

Figure 5: The architecture of the **Minimum Circuit** [10].

column, the time for the Xeon microprocessor in the second column, the time for the FPGA in the third column, and the speed-up factor in the right column. The speed-up factors range from 39.9 times for $n = 2$ to 62,111 times for $n = 8$.

For $n \geq 6$, the computation times for both the microprocessor and FPGA were too large to run to completion. In the case of the FPGA, we can calculate an accurate running time. In the case of the microprocessor, we ran the program over a subset of all functions and then prorated to obtain the time it would take to enumerate all functions.

Do we benefit from a speed-up when neither the FPGA nor the microprocessor can enumerate all functions? Indeed, the answer is Yes. For example, Rothaus [7] showed that all bent functions are contained in a set of functions whose degree is $\frac{n}{2}$ or less. Since the cardinality of this set is much less than the set of all 6-variable functions, it *is* possible to enumerate all 6-variable bent functions by the FPGA. Indeed, by enumerating only functions with degrees 3 or less, it required 5.7 minutes on an FPGA. However, it would have taken 27 days on the microprocessor, which is 6,805.9 times longer, as indicated in Table 1. We also achieved the 62,111 speed-up associated with $n = 8$ variable functions. In this case, we enumerated all 8-variable rotation symmetric functions (c.f. Cusick and P. Stănică [2]) and showed that the distribution of such functions is similar to the distribution of *all* functions with 4 and 5 variables [10].

Table 1: Speed-up obtained by the SRC-6 reconfigurable computer [10]

n	PC Compute Time (@2.8 GHz.)	SRC-6 Compute Time (@100 MHz)	Speed-up Factor
2	6.38 μsec.	0.16 μsec.	39.9 $\times$
3	457.0 μsec.	2.56 μsec.	178.5 $\times$
4	0.388 sec.	655.4 μsec.	592.0 $\times$
5	25.338 hours	42.9 sec.	2,126.3 $\times$
6	39,807,788 years	5,840 years	6,805.9 $\times$
7	2.05×10^{27} years	1.08×10^{23} years	19,005 $\times$
8	2.28×10^{66} years	3.67×10^{61} years	62,111 $\times$

4 Improving the Throughput

Although the FPGA has a much slower clock than the microprocessor, its speed of computation is much faster because of parallelism. For example, the many adders in the **Ones Count** circuit operate simultaneously in an FPGA, while these additions must be performed in serial on the microprocessor. However, in spite of this improvement, the throughput of the FPGA can be improved further. We describe this here. Unlike the previously discussed circuit, the following circuit has yet to be implemented.

One way to improve throughput is to simply replicate the circuit and simultaneously apply different functions for testing. An inefficiency exists because of many unneeded tests of the distance to an affine function. That is, it is sufficient to declare a function as non-bent if it fails one distance test. Yet, in all the implementations discussed so far, for every prospective bent function, all distances are computed.

To achieve greater throughput, we propose the **circular pipeline**. In its simplest form, there are 2^n stages, where each stage tests the resident function against a linear function. If the resident function is not a bent distance (either of $2^{n-1} \pm 2^{n/2-1}$) to the linear function, it is discarded and replaced by another function. If the resident function is a bent distance, it moves onto the next stage. A function is bent if it passes through all stages. Note that it is unnecessary to test against all affine functions; a function that is a bent distance from all linear functions is a bent function.

Fig. 6 shows the architecture of the circular pipeline. On the left, 2^n function truth tables are clocked into register *In* through a MUX. On the right, i truth tables leave this register and are inserted into the circular pipeline, where $0 \leq i \leq 2^n$. We assume that any or all of the truth tables from *In* leave and are inserted into the circular pipeline. Those truth tables that are not inserted into the circular pipeline are inserted into register *Reservoir*. This process is special. Register *Reservoir* is similar to a shift register. However, it has an interconnection matrix, shown as *Switch Network*, that assures there are no "gaps"

caused by elements of register *In* that went into the circular pipeline. Register *Reservoir* stores 2^{n+1} truth tables. These are truth tables that failed to go into the *Circular Pipeline* because of a truth table that passed one test in the *Circular Pipeline*. The *Switch Network* assures that the bottom register is fully occupied with truth tables when it is applied to register *In*. When this is applied to *In*, then the rest of register *Reservoir* is moved down.

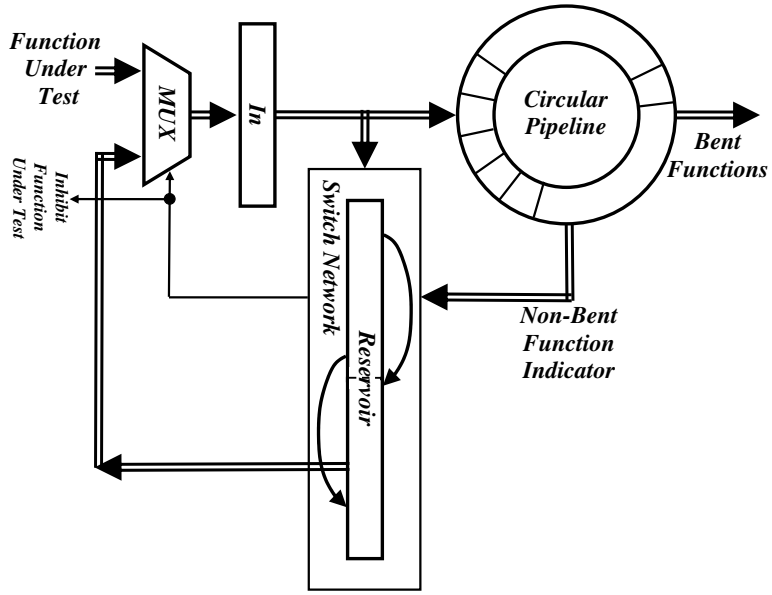

Figure 6: Architecture of the circular pipeline

This circuit is significantly more efficient than any we have proposed previously. This is because any function that fails a test is ejected immediately. However, its efficiency depends on the extent to which non-bent functions fail a test for bentness. For example, this method would not be so efficient if most non-bent functions had many bent distances to linear functions. Conversely, this method would be very efficient if only bent functions were a bent distance away from linear functions. Indeed, in this case, it would be necessary only to test whether a function was a bent distance from any linear function. This presents us with a very interesting question: "How long will an non-bent function persist in the circular pipeline?"

A partial answer was derived by simulating the circular pipeline for 4-variable functions. In this experiment, the truth tables were applied lexicographically, as $(0,0,\ldots,0,0,0)$, $(0,0,\ldots,0,0,1)$, $(0,0,\ldots,0,1,0)$, ..., $(1,1,\ldots,1,1,1)$, to a software simulation of the circular pipeline. From this, we computed the distribution of clock periods that a function survived the pipeline. Fig. 7 shows the distribution of functions to the time spent in the pipeline. For example, 49,698 of the 65,536 or 76% of the 4-variable functions stayed in the pipeline for only one clock period. 8,130 or 12% stayed in the pipeline for only two clock periods. 896 functions stayed in the pipeline for 16 clock periods. This last observation is an expected result, since 896 functions are bent. The number of times a function stays in the pipeline is called its **persistence**. The average persistence is surprisingly small, 1.65. Indeed, from the data shown in Fig. 7, we can declare a function to be bent if it stays in the pipeline for 10 or more clock periods. That is, no non-bent function persists in the pipeline for 10 or more clock periods. Knowing this would save some clock periods overall, since 896 of the 65,536 functions would have to stay in the pipeline only 10 clock periods instead of 16. We are interested in analyzing the throughput of the circular pipeline.

Definition 4.8. *The throughput T_n of a circular pipeline for n-variable bent functions is*

$$T_n = \frac{2^n}{P_{\text{avg}}}, \tag{1}$$

where $P_{\text{avg}} = \frac{1}{2^{2^n}} \sum_{i=0}^{2^{2^n}-1} P(f_i)$ is the average persistence, for $P(f_i)$, the persistence of function f_i.

We know that $1 \leq T_n \leq 2^n$. The lower bound occurs if all functions on n-variables are bent. The upper bound occurs if every function is non-bent (and thus, fails its test immediately). For example, if

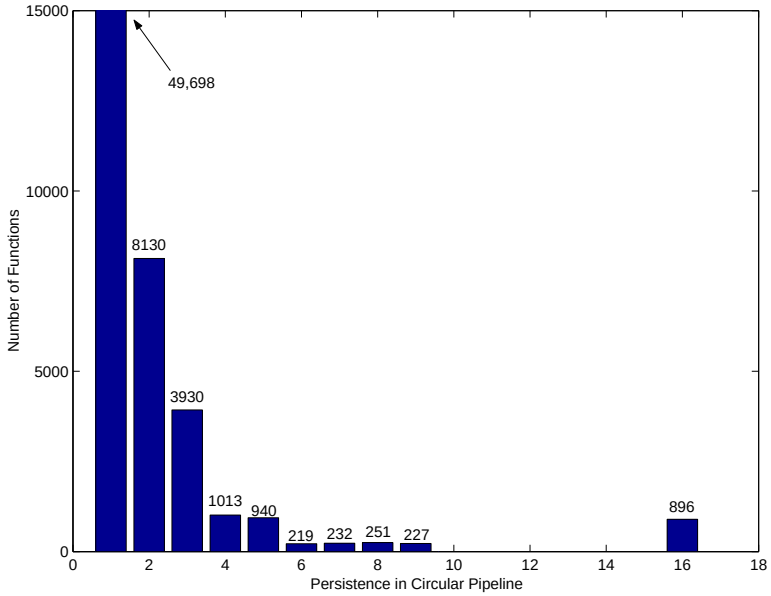

Figure 7: Distribution of Functions to Persistence in a Circular Pipeline for $n = 4$

$P_{\text{avg}} = 1$ (every function stays in the pipeline for exactly one clock period), then $T_n = 2^n$, because, at every clock all 2^n functions in the pipeline are replaced by 2^n new functions. This continues until there are no more functions. Thus, in the sieve, 2^n functions are examined every clock period.

From this, we can say, for the 4-variable example, that $T_4 = 2^2/1.65 = 9.7$. That is, with the circular pipeline, we can expect a throughput of nearly 10 times that of the circuit shown in Section 3. Note that, in this calculation, we do *not* take advantage of the fact that a function can be declared bent and removed from the pipeline after it has passed 10 stages.

If a function has zero bent weights, its persistence in Fig. 7 is 1. However, a function with one or more bent weights may also contribute to the functions with persistence 1. This is because such a function was placed in the circular pipeline in a stage that tested it against a linear function which was not a bent distance away. From this we conclude that the distribution of functions to persistence may be different depending on the order in which functions are placed into the circular pipeline. Therefore, we seek a persistence measure that is independent of this order.

Definition 4.9. *An n-variable function is said to have a* **bent weight** *if at least one distance to an affine function is $2^{n-1} \pm 2^{n/2-1}$ (the bent distance).*

Example 4.7. *An affine function has no bent weights. A bent function has 2^{n+1} bent weights, the maximum.* *(End of Example)*

Fig. 8 shows the distribution of 4-variable functions to the number of bent weights. For example, the bar on the extreme left shows that 33,920 functions have no bent weights, while the bar on the extreme right shows that 896 functions have 16 bent weights. The latter are the bent functions. Near the middle are two bars. These show that 3,840 functions have seven bent weights, while 26,880 have eight bent weights. For such functions, the distances to affine functions are divided nearly in half between bent distances and non-bent distances. Assuming these functions enter the circular pipeline randomly, we expect that approximately half of the time they will pass to the next stage and half of the time they will be rejected. Note that all functions with 0 bent weights stay in the pipeline for one clock period. If a functions has 1 bent weight, it stays in the pipeline for either one or two clock periods. Similarly, functions with 8 bent weights stay in the pipeline for one, two, ... , or nine clock periods. Thus, we see in Fig. 7 a small number of functions (227) with persistence 9. For such a function, the highly unlikely case occurred in which all the eight stages through which it passed corresponded to the linear functions from which it was a bent distance away. Bent functions stay in the pipeline for 16 clock periods. They are ejected after 16 stages (not 17) even though they pass the last bent test.

It is interesting to note that the histogram of Fig. 7 shows a rise in the number of functions with persistence 6 through 9. This is likely due to the concentration of 4-variable functions with 7 or 8 bent weights. Indeed, there are three (local) peaks in the number of functions with various persistence values

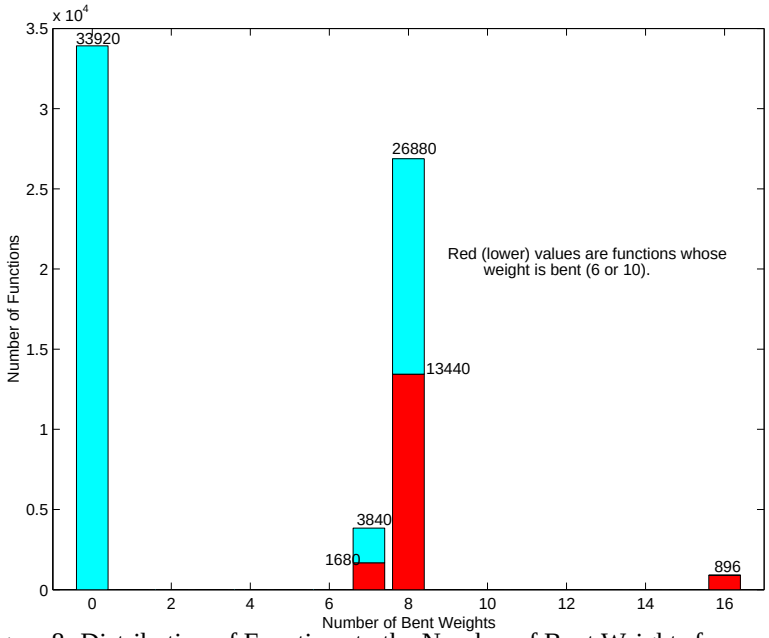

Figure 8: Distribution of Functions to the Number of Bent Weights for $n = 4$

(in Fig. 7) corresponding to the three peaks in the number of functions with various bent weights (in Fig. 8), near 0, 6-9, and 16.

The results of Fig. 8 beg the question of the bent weights of $n > 4$. For example, does the large number of functions with 0 bent weights also occur for $n > 4$? What form does the distribution of functions to the number of bent weights take for other non-bent functions?

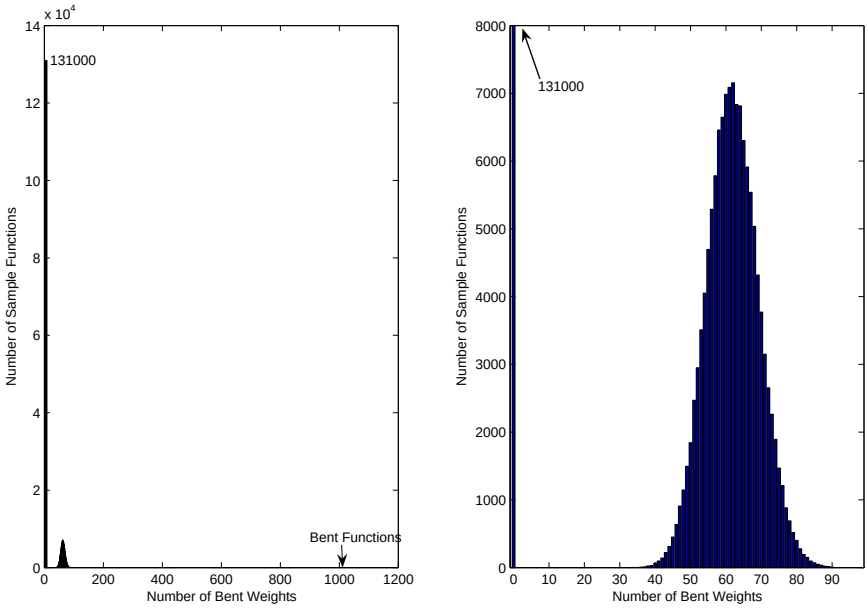

Figure 9: Distribution of Functions to the Number of Bent Weights for $n = 10$ (262,144 samples)

To answer these questions, we repeated the analysis on 4-variable functions on 10-variable functions. It is impossible to enumerate all n-variable functions for $n \geq 6$ because of excessive computation time. However, one can enumerate random n-variable functions. Doing this for $n = 10$ yields the plot of Fig. 9. The left plot shows the full plot. There are a large number of functions, 131,000, with no bent weights (i.e. about half of all samples). This is shown as a single line on the extreme left. Just to its right is a small

group of functions where the number of bent weights is approximately 64. Bent functions have 1024 bent weights, and they would be on the right, as indicated by the arrow. Because the sample size, 131,000, is so small compared to the total number of functions, 2^{1024}, no bent functions were enumerated. Indeed, the largest number of bent weights in any non-bent function was found to be 151. The right plot is an expanded version of the left plot. It shows clearly a distribution of functions centered around 64 bent weights. This plot shows that the majority of the functions would drop out of the circular pipeline after 90 clock periods. In the case of $n = 10$, the circular pipeline is 1024 stages long. Indeed, since functions with approximately 64 bent weights, and 64 is only one-sixteenth of 1024, most functions will drop out well before 90 clock periods. In the case of $n = 10$, the bent weights are 496 and 528.

Lemma 4.1. *Let f and g be two functions in the same affine class (i.e. there exists an affine function h, such that $f = g \oplus h$). Then, f and g have the same number of bent weights.*

From Lemma 4.1, if f has 2^n bent weights, then f is bent and so are all functions in the same affine class as f. One seeks then an affine class in which all functions have many bent weights that also contains a balanced function.

This suggests a new kind of nonlinearity. Functions that have 8 bent weights in Fig. 8 are closer to bent functions than any other functions. Indeed, it can be seen, that a substantial number of these functions are balanced; i.e. 10,080 of them have 8 1's and 8 0's. It is also interesting to note that such functions can have nonlinearity no greater than 4, since the smallest weight is 4 among the functions that have 8 bent weights. Among 4-variable functions, the largest nonlinearity a function can have and *not* be bent is 5.

5 Analytical Results

Lemma 5.2. *The number of functions $N(NL, w)$ with nonlinearity NL and weight w is*

$$N(NL, w) = \binom{2^n}{w}, \text{ for } 0 \le NL = w \le 2^{n-2}. \tag{2}$$

$$N(NL, w) = \binom{2^{n-1}}{p_0}\binom{2^{n-1}}{p_1}(2^{n+1} - 2), \quad \begin{array}{l}\text{where } w = 2^{n-1} - p_1 + p_0, \\ \text{for } 0 \le NL = p_0 + p_1 \le 2^{n-2} - 1.\end{array} \tag{3}$$

$$N(NL, w) = \binom{2^n}{2^n - w}, \text{ for } 0 \le NL = 2^n - w \le 2^{n-2}. \tag{4}$$

Proof: *For all but two exceptions, the functions described above are the smallest distance to a single affine function. For example, the function $f = x_1 x_2 \ldots x_n$ with one 1 entry is a distance 1 away from the constant 0 function (and a larger distance away from all other affine functions). The two exceptions correspond to (2) for $NL = w = 2^{n-2}$ and (4) for $NL = 2^n - w = 2^{n-2}$. Consider the first case only; the second is similar. This corresponds to functions that have weight 2^{n-2}, and so are a distance 2^{n-2} from the constant 0 function. Among these functions are some that are a distance 2^{n-2} from non-trivial affine functions, but at no greater a distance. It follows that the non-linearity of all functions of weight 2^{n-2} is 2^{n-2}.* ∎

Lemma 5.3. *A function has odd weight iff it has odd nonlinearity.*

Proof: *All affine functions have even weight. If f has even weight, all distances to affine functions are even. Thus, the smallest distance is even, and it follows that the nonlinearity of f is even. Conversely, if the weight of f is odd, then all distances to affine functions are odd. The smallest distance is odd, and so f has odd nonlinearity.* ∎

Lemma 5.4. *At least one-half of all functions have no bent weights.*

Proof: *Half of all functions have a truth table with an odd number of 1's. From the proof of Lemma 5.3, all distances between such a function and an affine function are odd. Since a bent weight is even, functions with odd weight have no bent weights.* ∎

Lemma 5.4 explains why more than half of all functions have no bent weights, as observed in Figs. 8 and 9.

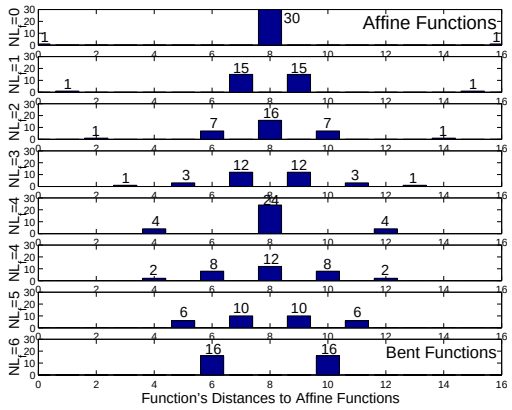

Figure 10: All distributions of functions to distances to affine functions for $n = 4$.

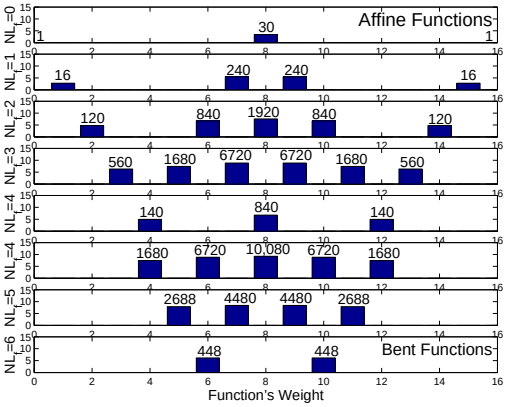

Figure 11: Distribution of functions to weights with the distance distributions in Fig. 10.

5.1 Distribution Among Distances to Affine Functions

We can gain insight into the distances between functions and affine functions, by considering the distribution of distances. Fig. 10 shows the distribution of distances to affine functions among 4-variable functions with various nonlinearities. For example, the top plot shows the distance distributions for affine functions. That is, for any affine function f, f is a distance 0 from an affine function (itself); it is a distance 16 from another affine function ($\bar{f}$); and, it is a distance 8 from all of the other 30 affine functions. In the case of a function f with nonlinearity 1, f is a distance 1 from an affine function, a distance 7 from 15 affine functions, a distance 9 from 15 affine functions, and a distance 15 from one affine function. This is shown in the second plot of Fig. 10. The last plot of Fig. 10 shows the bent functions. Here, each bent function is a distance 6 from 16 affine functions and a distance 10 from the other 16 affine functions. It is seen from Fig. 10 that, as the nonlinearity increases, the distribution of distances to affine functions narrows. This is expected, since nonlinearity is a measure of the *maximum* distance to affine functions.

Fig. 11 shows the distribution of functions with various nonlinearity values according to the function's weight. For example, the top plot of Fig. 11 shows the distribution of weights for all functions with nonlinearity 0, the affine functions. One has weight 0, the constant 0 function, one has weight 16, the constant 1 function, and 30 have weight 8.

It is interesting that, in Figs. 10 and 11, that there are *two* distributions for nonlinearity $NL_f = 4$. The first, where four functions are a distance 4 from affine functions, 24 are a distance 8, and four are a distance 12, correspond to products of two variables. For example, $f = x_1x_2$, where $TT_f = (0000000000001111)$ are a distance 4 from the four affine functions

$$
\begin{aligned}
g_1 &= 0(TT_{g_1} = (0000000000000000)),\\
g_2 &= x_1(TT_{g_2} = (0000000011111111)),\\
g_3 &= x_2(TT_{g_3} = (0000111100001111)), \text{ and}\\
g_4 &= 1 \oplus x_1 \oplus x_2(TT_{g_4} = (1111000000001111)) \ .
\end{aligned}
$$

Further, $f = x_1x_2$ is a distance 12 from the complement of these functions. And, it is a distance 8 from the remaining 24 affine functions. Note that this characterizes functions with this distribution of distances to affine functions. Namely, they are the AND of two nontrivial affine functions exclusive ORed with an affine function. The functions associated with the other distribution of distances corresponds to all other functions with weight 4 or the exclusive OR of those functions with an affine function.

Note that all 4-variable functions with nonlinearity $0 \leq NL \leq 3$ are the minimum distance from a *unique* affine function. All 4-variable functions with weight 4 have nonlinearity $NL = 4$. This same statement is *not* true of functions with nonlinearity $5 \leq NL \leq 6$. For $n = 5$, there are $\binom{16}{5} = 4,368$ functions with weight 5, but only 2,688 of them have nonlinearity 5. And, for $n = 6$, there are $\binom{16}{6} = 8,008$ functions with weight 6, but only 448 of them have nonlinearity 6. The functions with weight 5 or 6 but not nonlinearity 5 or 6, have a lower value for nonlinearity. For example, there are 2,688 functions with nonlinearity 5 and weight 5. We would compute this by observing that the only other nonlinearity of 5 and weight 5 occur for functions with nonlinearity 3. There are 1680 such functions. However, there are $\binom{16}{5} = 4368$ total with weight 5. Thus, the number of functions with weight 5 and nonlinearity 5

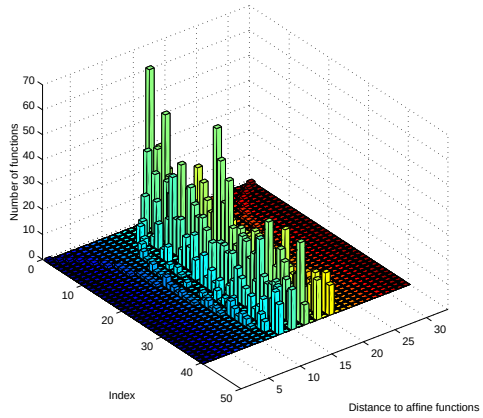

Figure 12: All distributions of functions to distances to affine functions for $n = 5$.

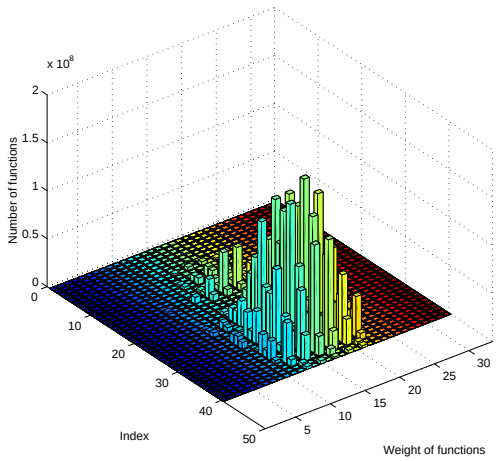

Figure 13: Distribution of functions to weights with the distance distributions in Fig. 12.

must be $\binom{16}{5} - 1680 = 2688$. This gives one hope for computing the other values of nonlinearity and weight.

There are two sources of function with nonlinearity 4. For one, the function is a distance 4 from 4 affine functions, a distance 8 from 24 affine function, and a distance 12 from 4 affine functions. These functions have the form $g_1 g_2$, where g_1 and g_2 are distinct nontrivial affine functions that are not complements of each other. For example, for $g_1 = x_1$ and $g_2 = x_2$, the function $g_1 g_2 = x_1 x_2$ is a distance 4 from g_1, g_2, $1 \oplus g_1 \oplus g_2$, and the constant 0 function. It is a distance 12 from the complement of these four functions, all of which are also affine. And, it is a distance 8 from the other 24 affine functions. How many functions have this distribution of distances to the affine functions? Among the 30 nontrivial affine functions, there are $\binom{30}{2} - 15 = 420$ ways to choose distinct pairs of functions that are not complements of each other. However, this number triple counts the number of distinct functions. That is, $g_1 g_2$ is formed from g_1 ANDed with g_2. But, there are two other pairs of distinct nontrivial affine functions that are not complements of each other that can form $g_1 g_2$, namely the pairs (g_1 and $\bar{g}_1 \oplus g_2$) and ($g_1 \oplus \bar{g}_2$ and g_2). Thus, the total number of functions with this distribution of distances to the affine functions are $420/3 = 140$.

The other set of functions with nonlinearity 4 have 1680 functions of weight 4 that are have nonlinearity 4, as shown in Fig. 11. There are $\binom{16}{4} - 140 = 1,680$ of these.

Fig. 12 shows the same data for $n = 5$ as Fig. 10 does for $n = 4$. Here, there are too many bars to label. But, the x-axis shows the distances a function can have to the 64 5-variable affine functions. The y-axis shows an index for the 40 possible distributions. And, the vertical axis shows the number of functions. Note that this is not the log of number of functions; instead is the actual number. The functions with high nonlinearity are in the front (to the right). Interestingly, there are two distributions with maximum nonlinearity, 11, one correspond to 3 distances and the other 5 distances. This is to be compared to the case of bent functions (and even number of variables), where there are always 2 distances.

Fig. 13 shows the distribution of weights of functions that have the distance distributions shown in Fig. 12. This shows that the functions are concentrated in the regions of higher nonlinearity and this number then drops off rapidly near the highest nonlinearity.

6 Concluding Remarks

We have demonstrated that there is a significant advantage to using a reconfigurable computer to sieve for bent functions. A speedup in computation time of more than 60,000 times is possible because of the large amount of logic that can be configured to an architecture ideally suited to the problem. For example, Figs. 4 and 5 show that many simultaneous additions and comparisons are needed. Although these are performed much more slowly on an FPGA than on an Intel processor chip, they can be replicated enough times that one function is tested every clock period of the FPGA's 100 MHz's clock. The code is easily scalable. The use of parameters in Verilog allows the number of variables to be specified in the top

module and then transmitted down to lower level modules.

It is likely that other problems are as well-suited to computation by reconfigurable computer. For example, [1] discusses three cryptographic properties, strict avalanche criterion [3], [4], propagation criterion, and correlation immunity [12], that potentially have efficient FPGA architectures. Still another cryptographic property, algebraic immunity [6] may also have efficient architectures.

For implementation information on bent function discovery by reconfigurable computer, see [8] and [9].

References

[1] J. T. Butler and T. Sasao, "Bent functions and their relation to switching circuit theory – A tutorial", *Proc. of the Reed-Muller Workshop 2009*, May 23-24, 2009, Naha, Okinawa, Japan, 127–136.

[2] T. W. Cusick and P. Stănică, *Cryptographic Boolean Functions and Applications*, Academic Press, San Diego, CA, 2009.

[3] R. Forré, "The strict avalanche criterion: Spectral properites of Boolean functions and an extended definition," *Advances in cryptology, Crypto'88* , pp. 450-468.

[4] K. J. Horadam, *Hadamard Matrices and Their Applications*, Princeton University Press, 2007.

[5] Y. Komamiya, *Theory of Computing Networks*, Research of ETL, Sept. 1959.

[6] W. Meier, E. Pasalic, and C. Carlet, "Algebraic attacks and decomposition of Boolean functions", *Advances in Cryptology-CRYPTO 2004 (Lecture Notes in Computer Science)*, Berlin, Germany: Springer-Verlag, 2003, vol. 3027, pp. 474-491.

[7] O. S. Rothaus, "On 'bent' functions", *J. Combinatorial Theory, Ser. A*, Vol. 20 (1976), 300–305.

[8] S. W. Schneider, "Finding bent functions using genetic algorithms", M.S. Thesis, Naval Postgraduate School, Monterey, CA, September 2009.

[9] J. L. Shafer, "An analysis of bent function properties using the transeunt triangle and the SRC-6 reconfigurable computer", M.S. Thesis, Naval Postgraduate School, Monterey, CA, September 2009.

[10] J. L. Shafer, S. Schneider, J. T. Butler, and P. Stanica, "Enumeration of bent Boolean functions by reconfigurable computer," *The 18th Annual International IEEE Symposium on Field-Programmable Custom Computing Machines*, Charlotte, NC, May 2-4, 2010, pp. 265-272.

[11] C. Shannon: "Communicaton theory in secrecy systems", *Bell Systems Technical Journal*, 1949, pp. 656 - 715.

[12] T. Siegenthaler, "Correlation immunity of nonlinear conbining functions for cryptographic applications," *IEEE Trans. on Information Theory*, IT-30(5), pp. 776-780, Sept. 1984.

[13] T. Sasao, *Switching Theory for Logic Synthesis*, Kluwer Academic Publishers, 1999.

[14] "SRC-6 Carte^{TM} v3.2 C Programming Environment Guide", SRC Computers, LLC, Nov. 22, 2009, p. 1.